Department I - C Plus Plus

Modern and Lucid C++
for Professional Programmers

Week 8 – STL Algorithms

Thomas Corbat / Felix Morgner

Rapperswil, 03.11.2020

HS2020

Cevelop

Your C++ deserves it

OST Ostschweizer Fachhochschule

IFS INSTITUTE FOR SOFTWARE

- **You know when and why to prefer standard algorithms over hand-written loops**

- **You can name the most important algorithms in the STL**

- **You can explain certain pittfalls when using STL algorithms**

- **You can explain the signature of a standard algorithm**

- **You can write programs that correctly use standard algorithms**

- **Recap Week 7**

- **Standard Algorithms**

  - Motivation

  - Algorithm Basics

  - Functors

  - More Algorithm Examples

  - Pitfalls
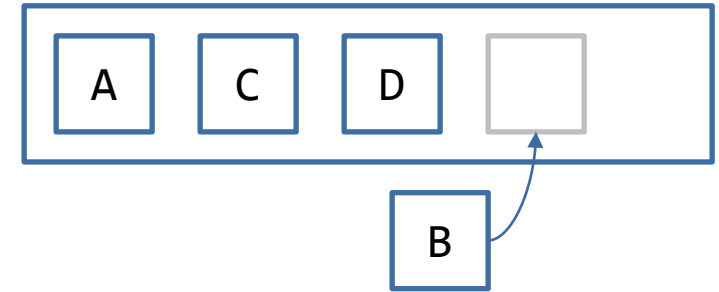
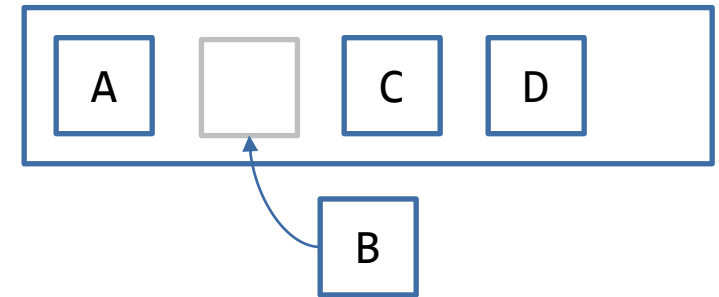- **Algorithm Headers**

Recap Week 7

- **Sequence Containers**

  - Elements are accessible in order as they were inserted/created

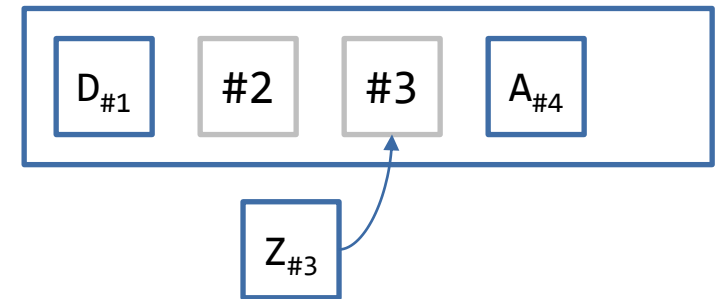 - Find in linear time through the algorithm find

A	C	D	

B

- **Associative Containers**

  - Elements are accessible in sorted order

  - find as member function in logarithmic time

A		C	D

B

- **Hashed Containers (Unordered Associative)**

  - Elements are accessible in unspecified order

  - find as member function in constant time

$D_{\#1}$	#2	#3	$A_{\#4}$

$Z_{\#3}$

```
┌─────────────────────────────────────────┐
│ ┌───────────────────────────────────┐   │
│ │ ┌───────────────────────────────┐ │   │
│ │ │         Input Iterator        │ │   │
│ │ └───────────────────────────────┘ │   │
│ │                                   │   │
│ │          Forward Iterator         │   │
│ └───────────────────────────────────┘   │
│                                         │
│            Bidirectional Iterator        │
│                                         │
│          Random Access Iterator          │
└─────────────────────────────────────────┘
```

```
┌───────────────────────────┐
│      Output Iterator       │
└───────────────────────────┘
```
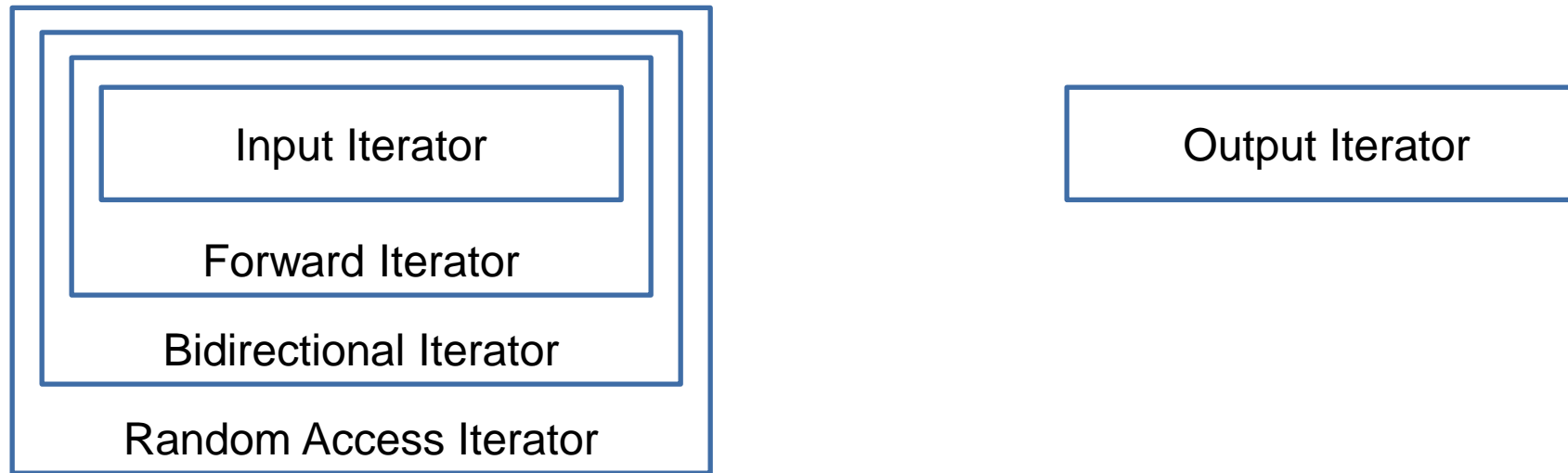
- **Different containers support iterators of different capabilities**

- **Categories are formed around increasing "power"**

    - `std::input_iterator` corresponds to istream_iterator's capabilities

    - `std::ostream_iterator` is an output_iterator

    - `std::vector<T>` provides random_access iterators

- **Declaring an iterator `const` would not allow modifying the iterator object**

  - ◼ You cannot call ++

- **`cbegin()` and `cend()` return const_iterators**

  - ◼ This does NOT imply the iterator to be `const`

  - ◼ The elements the iterator walks over are `const`

```cpp
std::vector<int> v{3, 1, 4, 1, 5, 9, 2, 6};

auto const iter1 = values.begin(); //std::vector<int>::iterator const
++iter1;

auto iter2 = values.cbegin(); //std::vector<int>::const_iterator
*iter = 2;
```

Why should we use algorithms?

● **How do you reverse all elements of a vector?**

```cpp
void reverse(std::vector<int> & values) {
  for (int i = 0; i <= values.size(); i++) {
    auto otherIndex{values.size() - i};
    auto tmp = values[0];
    values[i] = values[otherIndex];
    values[otherIndex] = values[i];
  }
}
```

- **How do you reverse all elements of a vector?**

```cpp
void reverse(std::vector<int> & values) {
  for (int i = 0; i <= values.size(); i++) {
    auto otherIndex{values.size() - i};
    auto tmp = values[0];
    values[i] = values[otherIndex];
    values[otherIndex] = values[i];
  }
}
```

- **Algorithms are less error-prone**

```cpp
void reverse(std::vector<int> & values) {
  reverse(begin(values), end(values));
}
```

- **What does the code do?**

```cpp
int ???????(std::vector<int> values) {
    int var = std::numeric_limits<int>::min();
    for (auto v : values) {
        if (v > var)
            var = v;
    }
    return var;
}
```

- **Algorithms can be much clearer regarding the programmer's intention**

```cpp
int ???????(std::vector<int> values) {
    if (values.empty()) {
        return std::numeric_limits<int>::min();
    }
    return *std::max_element(begin(values), end(values));
}
```

- **How many times is the condition evaluated?**

```cpp
bool contains(std::vector<int> values, int sought) {
  for (auto it = begin(values); it != end(values); it++) {
    if (*it == sought) return true;
  }
  return false;
}
```

- **And now?**

```cpp
bool contains(std::vector<int> values, int sought) {
  return find(begin(values), end(values), sought) != end(values);
}
```

- **To be fair: A modern compiler will optimize the condition in the `for` loop**

  - Nevertheless, an algorithm can be implemented to be more efficient than a hand-written loop

- **Correctness**

  ◾ It is much easier to use an algorithm correctly than implementing loops correctly

- **Readability**

  ◾ Applying the correct algorithm expresses your intention much better than a loop

  ◾ Someone else (or even you) will appreciate it when the code is readable and easily understandable

- **Performance**

  ◾ Algorithms might perform better than handwritten loops

    ◾ without sacrificing the "Readability" of your code

Algorithm Basics

- **`<algorithm>` header**

  - Filling

  - Finding

  - Property checking

  - Transformation

  - etc.

- **`<numeric>` header**

  - Generic numeric functions

  - Some functions can be applied in non-numeric contexts

- **Algorithms work with ranges specified by iterators**

- **Input**

  - 1 or 2 Ranges

  - Start
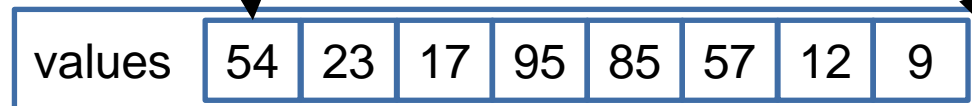
  - End (At least for first range)

- **Output**

  - Start

  - No end

- **Iterators specify ranges used for algorithms**

  - ▪ `First` – Iterator pointing to the first element in the range

  - ▪ `Last` – Iterator pointing beyond the last element in the range

  - ▪ If `First == Last` the range is empty

```
std::vector<int> values{54, 23, 17, 95, 85, 57, 12, 9};
std::xxx(begin(values), end(values), ...);
```

values | 54 | 23 | 17 | 95 | 85 | 57 | 12 | 9

- **Streams need a wrapper to be used with algorithms**

  - `std::ostream -> std::ostream_iterator<T>`

  - `std::istream -> std::istream_iterator<T>`

  - Default-constructed `std::istream_iterator<T>` marks EOF

```cpp
void redirect(std::istream & in, std::ostream & out) {
  using in_iter = std::istream_iterator<char>;
  using out_iter = std::ostream_iterator<char>;
  std::copy(in_iter{in}, in_iter{}, out_iter{out});
}
```

- **Executing an operation for each element in a range**

- **Operation is a function, lambda or functor**

  - ◾ Which is called for each element

- **Input iterators for range**

- **Returns the function/object passed as operation**

| counts | 3 | 0 | 1 | 4 | 0 | 2 |

f()
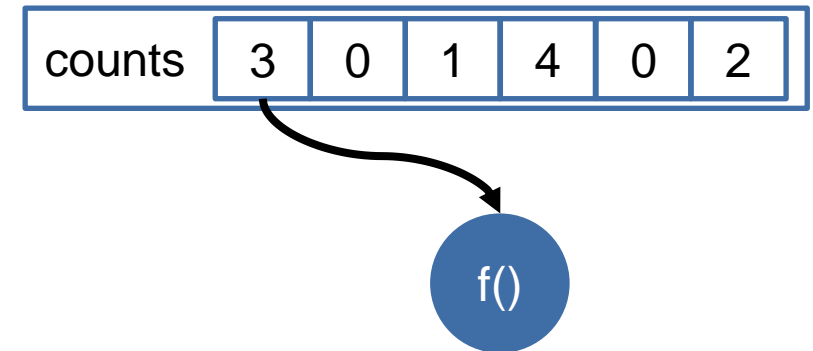
```cpp
std::vector<unsigned> values{3, 0, 1, 4, 0, 2};

auto f = [](unsigned v) {};
std::for_each(begin(values), end(values), f);
```

- **Signature of `std::for_each` from cppreference.com:**

Template Header		
template<class InputIt, class UnaryFunction>		
UnaryFunction	**for_each**(InputIt first, InputIt last, UnaryFunction f);	
Returntype	*Name*	*Parameters*

- **Each algorithm has**

  - A name

  - Parameters

  - A return type or is `void`

- **The description specifies the requirements**

- **Parameter and return types are usually template parameters (next week)**

- **Algorithms work with the iterator categories**

  - Input iterator

  - Forward iterator

  - Bidirectional iterator

  - Random access iterator

  - Output iterator

- **A functor is a type (class) that provides a call operator: `operator()`**

  - An object/instance of that type can be called like a function

  - It can provide multiple overloads of the call operator (usually not necessary)

  - Can take any number of parameters and have an arbitrary return type

- **Lambdas are functors internally**

```cpp
struct Accumulator {
  int count{0};
  int accumulatedValue{0};
  void operator()(int value) {
    count++;
    accumulatedValue += value;
  }
  int average() const;
  int sum() const;
};

int average(std::vector<int> values) {
  Accumulator acc{};
  for(auto v : values) { acc(v); }
  return acc.average();
}
```

```cpp
int average(std::vector<int> values) {
  Accumulator acc{};
  return std::for_each(begin(values), end(values), acc).average();
}
```
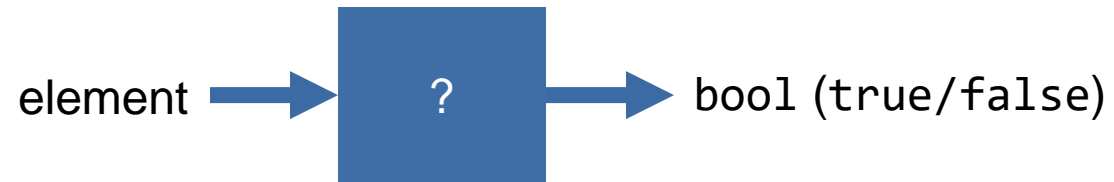
- **Predicate: A function or a lambda returning bool (or a type convertible to bool)**

  - For checking a criterion/condition

- **Unary Predicate**

  - One parameter

element ➡ ? ➡ bool (true/false)

```cpp
auto is_odd = [](int i) -> bool {return i % 2;};
```

- **Binary Predicate**

  - Two parameters

element 1 ➡
element 2 ➡ ? ➡ bool (true/false)

```cpp
auto divides = [](int i, int j) -> bool {return !(i % j);};
```

- **Lambdas make applying transform etc. quite easy**

```
transform(v.begin(),v.end(),v.begin(),[](int x){return -x;});
```

- **However, if there is only a simple expression used, there is a chance for reuse**

  ■ of the standard library's functor classes for almost all operators

```
transform(v.begin(),v.end(),v.begin(),std::negate<int>{});
```

  ■ also relational operators, e.g., sorting in descending order

```
sort(v.begin(),v.end(),std::greater<>{});
```

- **Binary arithmetic and logical**

  - ▪ `plus<> (+)`

  - ▪ `minus<> (-)`

  - ▪ `divides<> (/)`

  - ▪ `multiplies<> (*)`

  - ▪ `modulus<> (%)`

  - ▪ `logical_and<> (&&)`

  - ▪ `logical_or<> (||)`

- **unary**

  - ▪ `negate<> (-)`

  - ▪ `logical_not<> (!)`

- **binary comparison**

  - ▪ `less<> (<)`

  - ▪ `less_equal<> (<=)`

  - ▪ `equal_to<> (==)`

  - ▪ `greater_equal<> (>=)`

  - ▪ `greater<> (>)`

  - ▪ `not_equal_to<> (!=)`

```
transform(v.begin(),v.end(),v.begin(), v.begin(),std::multiplies<>{});
```

- **Associative containers allow an additional template argument for the comparison operation**

  - It must be a functor class giving a binary predicate

  - Predicate must be irreflexive and transitive

```
std::set<int, std::greater<>> reverse_int_set{};
```

- **Own functors can provide special sort order, e.g., case-less comparison of strings**

  - Caution: requirements for sorting must be fulfilled,

    - e.g. >= (std::greater_equal) is not allowed (is reflexive!)

```cpp
#include <set>
#include <algorithm>
#include <cctype>
#include <iterator>
#include <iostream>
struct caseless {
  using string = std::string;
  bool operator()(string const & l, string const & r) const {
    return std::lexicographical_compare(
                l.begin(), l.end(), r.begin(), r.end(),
                [](char l, char r){
                    return std::tolower(l) < std::tolower(r);
                });
  }
};
int main(){
  using std::string;
  using caseless_set = std::multiset<string, caseless>;
  using in = std::istream_iterator<string>;
  caseless_set const wlist{in{std::cin},in{}};
  std::ostream_iterator<string> out{std::cout,"\n"};
  copy(wlist.begin(), wlist.end(), out);
}
```

Binary Predicate

Strings as Ranges

Lambda as Binary Predicate on char

Pass Predicate Functor as Template Argument

- **Mapping one range to new values**

  - ■ Or two ranges of same size

- **Lambda/function/functor for map operation**

- **Input and output types can be different**

  - ■ As long as the operation has the correct types

counts	3	0	1	4	0	2

letters	'g'	'a'	'u'	'y'	'f'	'o'

f()

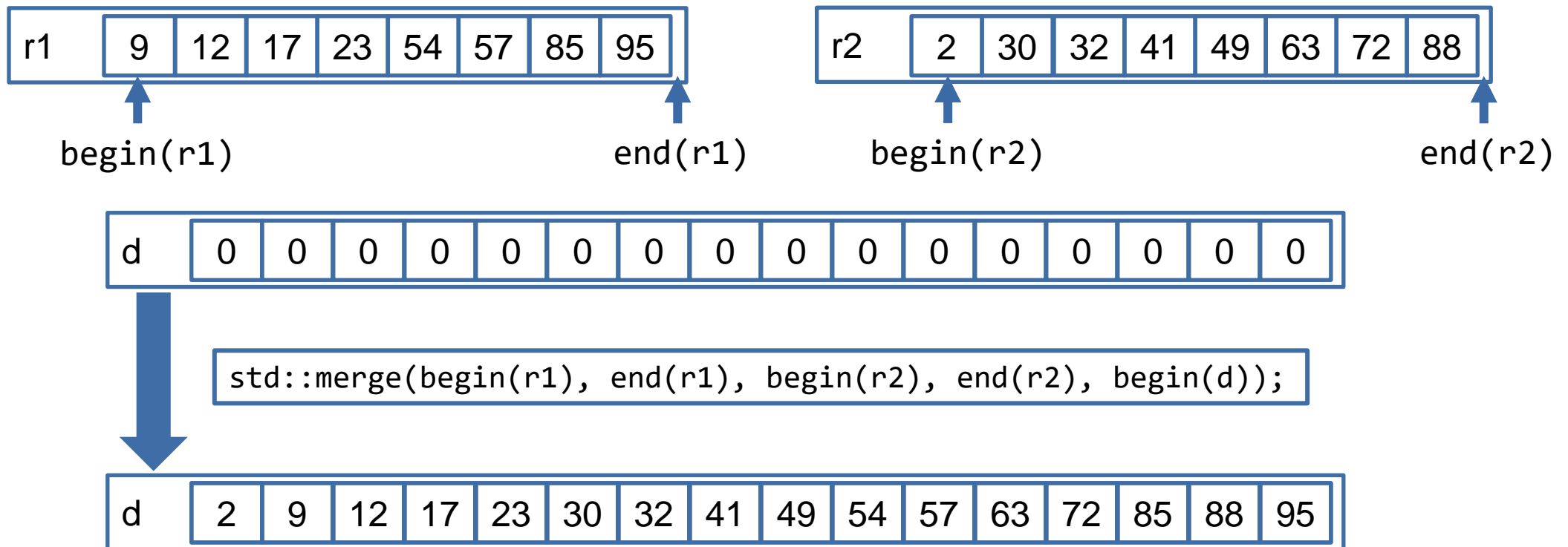| combined	"ggg"	""	"u"	"yyyy"	""	"oo"
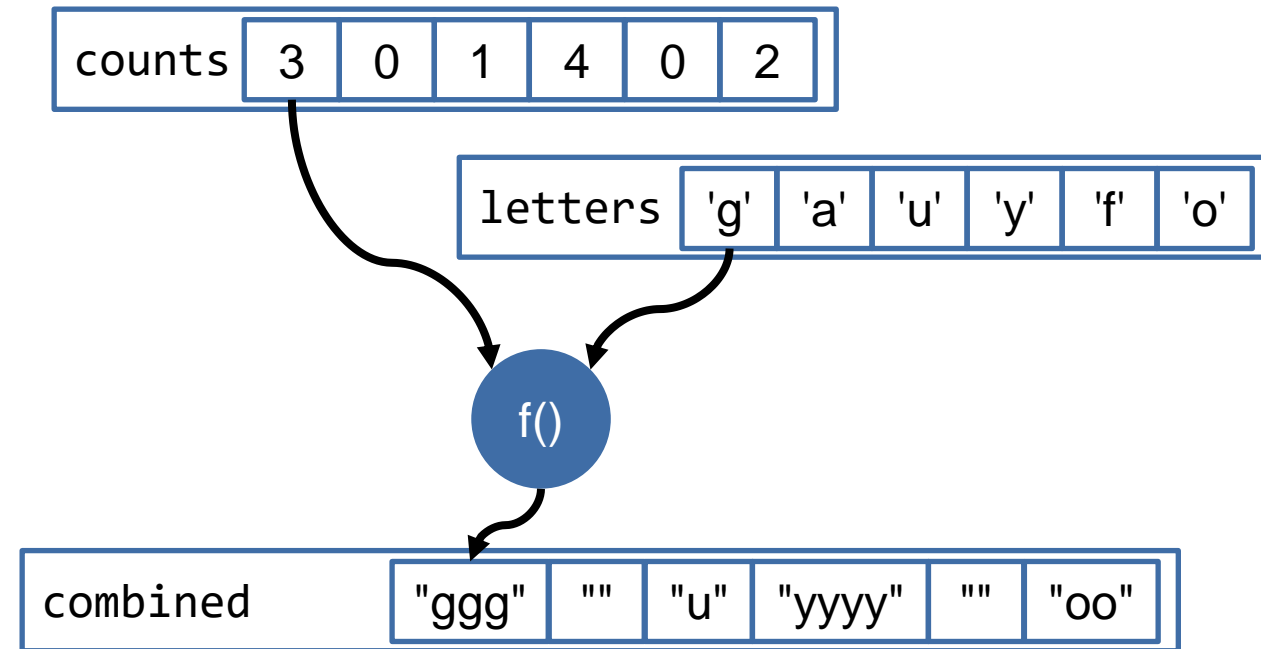
```cpp
std::vector<int> counts{3, 0, 1, 4, 0, 2};
std::vector<char> letters{'g', 'a', 'u', 'y', 'f', 'o'};
std::vector<std::string> combined{};
auto times = [](int i, char c) {return std::string(i, c);};
std::transform(begin(counts), end(counts), begin(letters),
               std::back_inserter(combined), times);
```

● **Merging two SORTED (!) ranges**

```cpp
std::vector<int> r1{9, 12, 17, 23, 54, 57, 85, 95};
std::vector<int> r2{2, 30, 32, 41, 49, 63, 72, 88};
std::vector<int> d(r1.size() + r2.size(), 0);
```

r1	9	12	17	23	54	57	85	95

begin(r1)   end(r1)

r2	2	30	32	41	49	63	72	88

begin(r2)   end(r2)

d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

```cpp
std::merge(begin(r1), end(r1), begin(r2), end(r2), begin(d));
```

d	2	9	12	17	23	30	32	41	49	54	57	63	72	85	88	95

?

```cpp
std::ostream_iterator<int> out{std::cout, ","};
int main() {
  std::set odd{1, 5, 7, 3, 9};
  std::set even{2, 8, 6, 4, 10};
  std::vector<int> all{};
  merge(begin(odd), end(odd),
        begin(even), end(even),
        std::back_inserter(all));
  copy(begin(all), end(all), out);
}
```

Correct

merge requires sorted ranges. However, the ranges in a set are sorted.

```cpp
int const minusIndex = 10;

auto splitDigits(int number) {
  std::string digits = std::to_string(number);
  std::vector<unsigned> indices(digits.size());
  transform(begin(digits), end(digits),
            begin(indices),
            [](char c) {
    return c == '-' ? minusIndex : c - '0';
  });
  return indices;
}
```
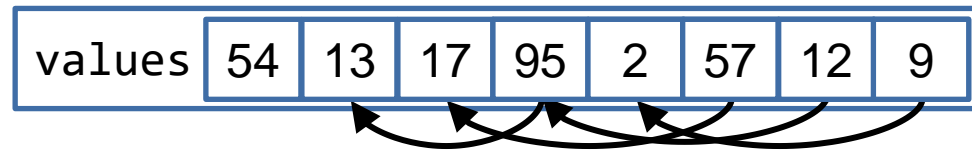
Correct

As long as the vector indices has enough elements, which can be overwritten, this code is correct.
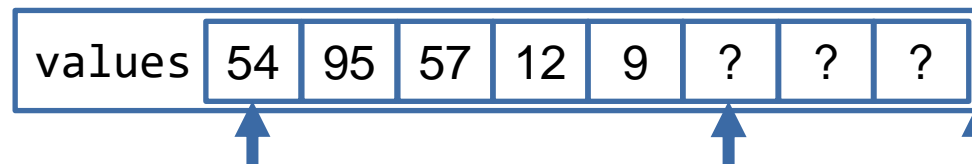
- **`std::remove` does NOT actually remove the elements**

  - ■ It moves the "not-removed" elements to the front and returns an iterator to the end of the "new" range

values	54	13	17	95	2	57	12	9

```
std::vector<unsigned> values{54, 13, 17, 95, 2, 57, 12, 9};
auto is_prime = [](unsigned u) {/*...*/};
auto removed = std::remove_if(begin(values), end(values), is_prime);
```
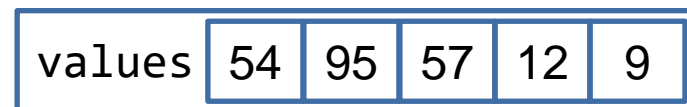
values	54	95	57	12	9	?	?	?

begin(values)   removed end(values)

- **To get rid of the "removed" elements usually the "erase" member function is called**

```
values.erase(removed, values.end());
```
➡

values	54	95	57	12	9

- **Some numeric algorithms can be used in non-numeric contexts**

- **Example `std::accumulate`**

  - Sums elements that are addable (+ operator)

  - Or based on a custom binary function

  - Returns the "sum" (accumulated value)

longMonths	Jan	Mar	May	Jul	Aug	Oct	Dec
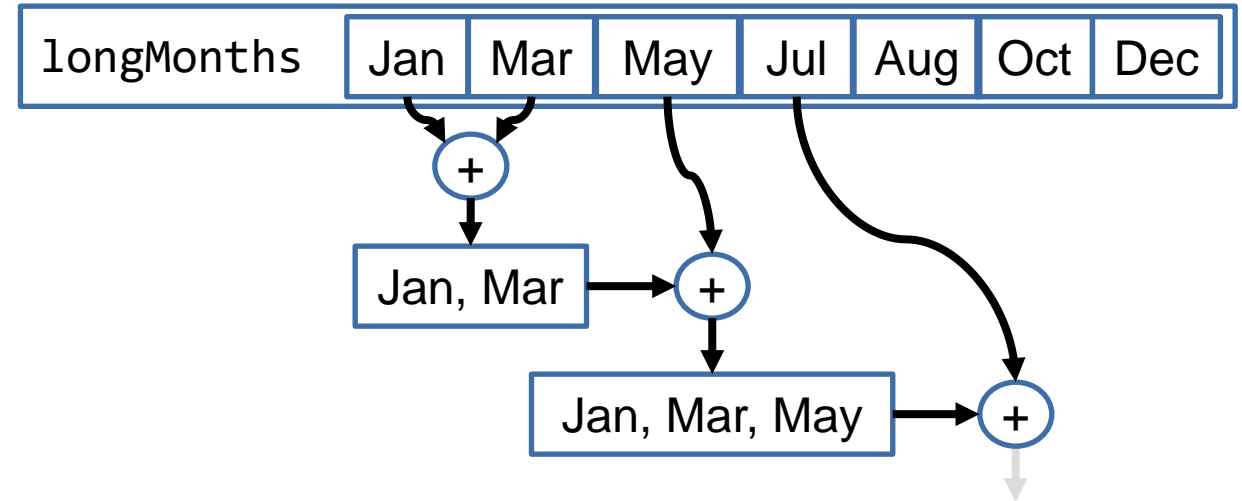
```
std::vector<std::string> longMonths{"Jan", "Mar", "May", "Jul", "Aug", "Oct", "Dec"};
std::string accumulatedString = std::accumulate(
        next(begin(longMonths)), //Second element
        end(longMonths),         //End
        longMonths.at(0),        //First element, usually the neutral element
        [](std::string const & acc, std::string const & element) {
                return acc + ", " + element;
        }); //Jan, Mar, May, Jul, Aug, Oct, Dec
```

- **Some algorithms have a variation with the _if suffix**

  - They take a predicate (instead of a value) to provide a condition

```cpp
std::set<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto isPrime = [](unsigned u) {/*...*/};
auto nOfPrimes = std::count_if(begin(numbers), end(numbers), isPrime);
```

- **Algorithms with the _if suffix**

  - count_if
  - find_if
  - find_if_not

  - copy_if
  - remove_if
  - remove_copy_if

  - replace_if
  - replace_copy_if

- **Some algorithms have a variation with the _n suffix**

  ■ The _n suffix is related to a number provided (usually instead of the "last" iterator)

```
std::set<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<unsigned> top5(5);
std::copy_n(rbegin(numbers), 5, begin(top5));
```

- **Algorithms with the _n suffix**

  ■ search_n

  ■ copy_n

  ■ fill_n

  ■ generate_n

  ■ for_each_n
  (Not implemented in GCC yet)

- **Implementing a binary heap on a sequenced container**

- **Requires random access iterator**

- **Guarantees:**

  - Top element is the largest (max)

  - Adding and removing elements have performance guarantees

- **Used for implementing priority queues**

- **Heap operations**

  - `make_heap` (3 * N comparisons)

  - `pop_heap` (2 * log(N) comparisons)

  - `push_heap` (log(N) comparisons)

  - `sort_heap` (N*log(N) comparisons)

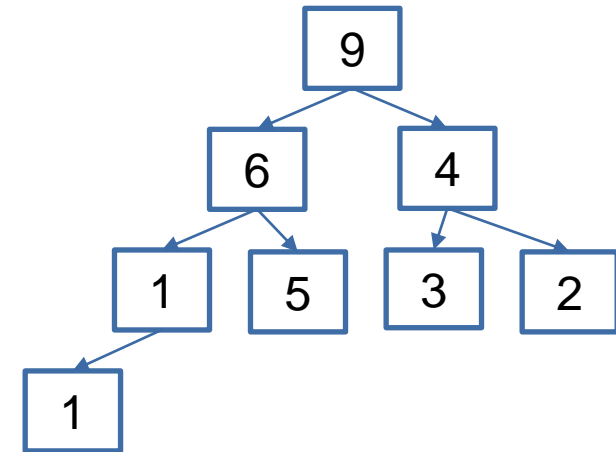| v		3	1	4	1	5	9	2	6	

```cpp
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
v.push_back(8);
push_heap(v.begin(),v.end());
sort_heap(v.begin(),v.end());
```

v | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6

v | 9 | 6 | 4 | 1 | 5 | 3 | 2 | 1

```cpp
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
v.push_back(8);
push_heap(v.begin(),v.end());
sort_heap(v.begin(),v.end());
```
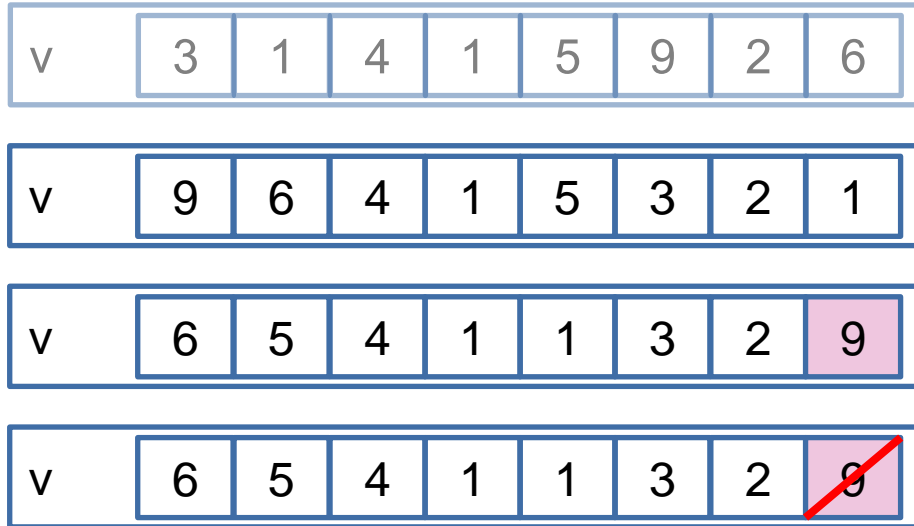
v	3	1	4	1	5	9	2	6

v	9	6	4	1	5	3	2	1

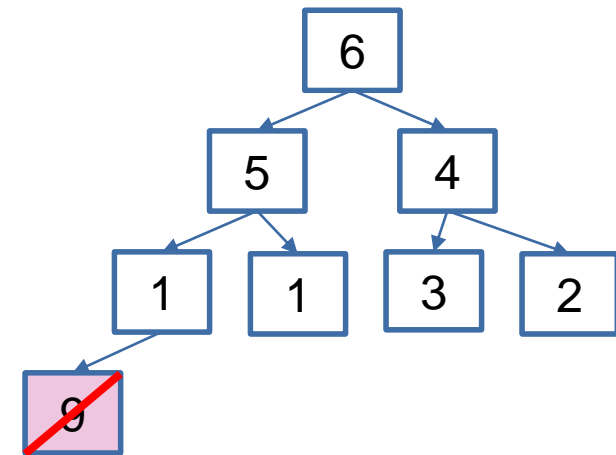| v	6	5	4	1	1	3	2	9

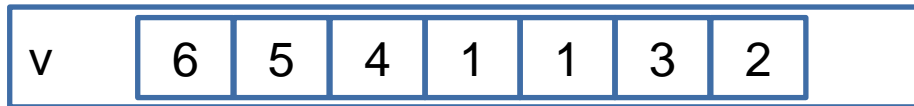| v	6	5	4	1	1	3	2	~~9~~

```
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
v.push_back(8);
push_heap(v.begin(),v.end());
sort_heap(v.begin(),v.end());
```

```
        6
       / \
      5   4
     / \  / \
    1  1 3  2
   /
  9
```

v | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6

v | 9 | 6 | 4 | 1 | 5 | 3 | 2 | 1

v | 6 | 5 | 4 | 1 | 1 | 3 | 2 | 9

v | 6 | 5 | 4 | 1 | 1 | 3 | 2 |

v | 6 | 5 | 4 | 1 | 1 | 3 | 2 | 8
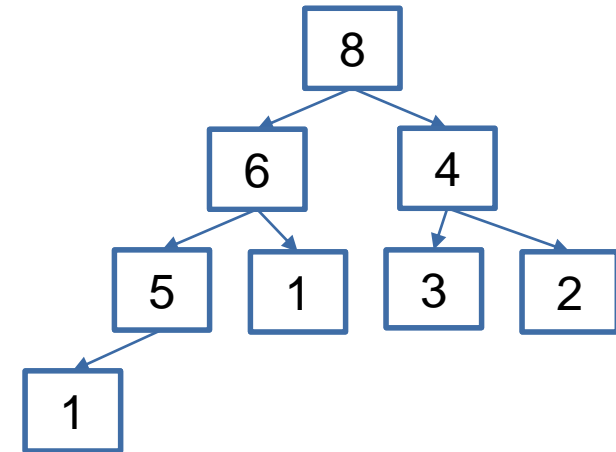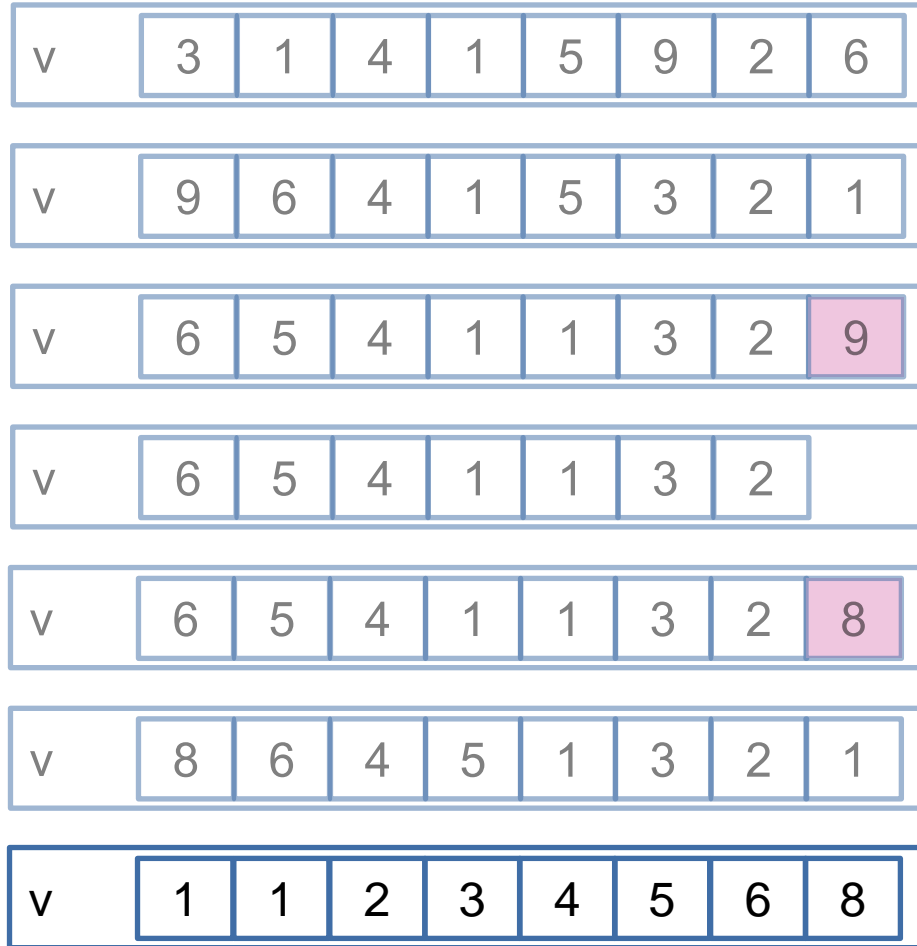
v | 8 | 6 | 4 | 5 | 1 | 3 | 2 | 1

```
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
v.push_back(8);
push_heap(v.begin(),v.end());
sort_heap(v.begin(),v.end());
```

| v | | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 |

| v | | 9 | 6 | 4 | 1 | 5 | 3 | 2 | 1 |

| v | | 6 | 5 | 4 | 1 | 1 | 3 | 2 | 9 |

| v | | 6 | 5 | 4 | 1 | 1 | 3 | 2 | |

| v | | 6 | 5 | 4 | 1 | 1 | 3 | 2 | 8 |

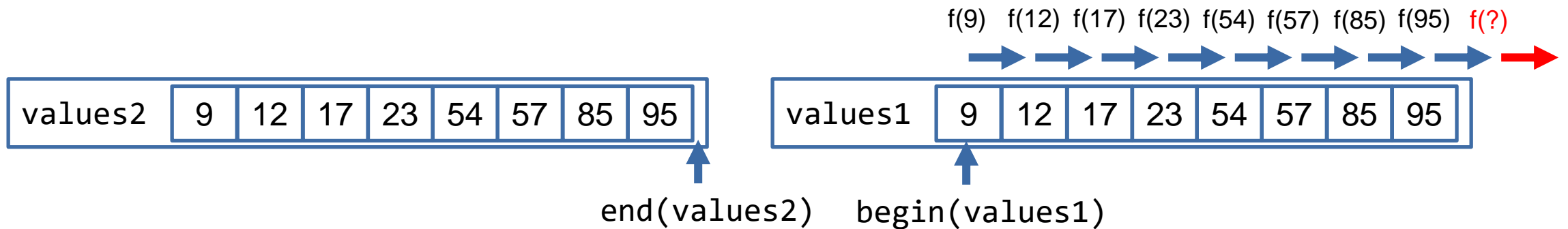| v | | 8 | 6 | 4 | 5 | 1 | 3 | 2 | 1 |

| v | | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 8 |

```
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
v.push_back(8);
push_heap(v.begin(),v.end());
sort_heap(v.begin(),v.end());
```

Pitfalls

- **It is mandatory that the iterators specifying a range need to belong to the same range**

- **Advancing the "first" iterator has to reach the "last" iterator eventually (without leaving the range)**

- **Otherwise, access of the value might result in undefined behavior**

f(9)   f(12)   f(17)   f(23)   f(54)   f(57)   f(85)   f(95)   f(?)

values2	9	12	17	23	54	57	85	95

values1	9	12	17	23	54	57	85	95

end(values2)   begin(values1)

```cpp
std::vector<unsigned> values1 = create_vector();
std::vector<unsigned> values2 = create_vector();

auto f = [](unsigned u) {/*...*/};
std::for_each(begin(values1), end(values2), f);
```

DANGER

**Undefined Behavior**

- **If you use an iterator for specifying the output of an algorithm you, need to make sure that enough space is allocated**

```cpp
std::set<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<unsigned> primes{};
auto isPrime = [](unsigned u) {/*...*/};
std::copy_if(begin(numbers), end(numbers), begin(primes), isPrime);
```

- **Vector `primes` is empty, which will not be changed by the iterator passed to copy**

| numbers | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- **The output is copied into (possibly) unallocated memory**

| primes | | 2 | 3 | 5 | 7 |

Unallocated memory

DANGER

Undefined Behavior
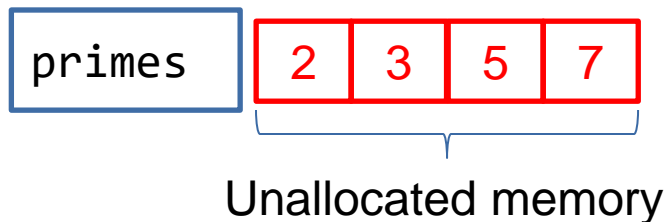
● **What if we want to insert elements into a container without pre-allocating the required memory?**

● **There are three inserter functions for wrapping containers that take care of this case**

- `back_inserter`

  - Creates an `std::back_insert_iterator`, which uses the `push_back` member function of the container

- `front_inserter`

  - Creates an `std::front_insert_iterator`, which uses the `push_front` member function of the container

- `inserter`

  - Creates an `std::insert_iterator`, which uses the insert member function of the container

```cpp
std::set<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<unsigned> primes{};
auto is_prime = [](unsigned u) {/*...*/};
std::copy_if(begin(numbers), end(numbers), back_inserter(primes), is_prime);
```

- **Some operations on containers invalidate its iterators**

- **Example:**

  values | 3 | 0 | 1 | 4 | 0 | 2

  - `std::vector<T>::push_back`

    > If the new size() is greater than capacity() then all iterators and references (including the past-the-end iterator) are invalidated. Otherwise only the past-the-end iterator is invalidated. [1]

    ```
    std::vector<unsigned> values{3, 0, 1, 4, 0, 2};

    auto f = [&values](unsigned v) {
      values.push_back(v);
    };

    std::for_each(begin(values), end(values), f);
    ```

f()

DANGER

Undefined Behavior
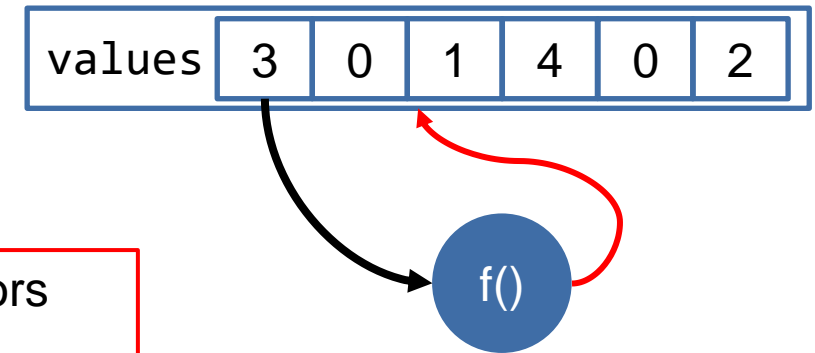
[1] http://en.cpppreference.com/w/cpp/container/vector/push_back

?

```cpp
bool isPrime(int value);

auto primes(std::vector<unsigned> const & values) {
  std::vector<unsigned> primeValues{};
  copy_if(begin(values), end(values),
          back_inserter(primeValues),
          isPrime);
  return primeValues;
}

int main() {
  std::istream_iterator<unsigned> inIter{std::cin};
  std::istream_iterator<unsigned> eof{};
  std::vector<unsigned> const values{inIter, eof};
  std::ostream_iterator<unsigned> out{std::cout};
  copy(begin(primes(values)), end(primes(values)), out);
}
```

Incorrect

Because primes returns a vector by value, the range in main is specified over two different instances of vector. The resulting behavior is undefined. Possibly, nothing is printed at all or the copy algorithm might continue copying the memory to the output as it never reaches end.

Algorithm Headers

- **Algorithms header**

  - Non-modifying sequence operations

  - Mutating sequence operations

  - Sorting and related operations

  - C library algorithms

- **Numerics header**

  - `accumulate`

  - `inner_product`

  - `partial_sum`

  - `adjacent_difference`

  - `iota`

- **all_of**

- **any_of** — condition checking

- **none_of**

- **find, find_if, find_if_not**

- **find_end**

- **find_first_of**

- **adjacent_find** — finding elements

- **count, count_if**

- **search, search_n**

- **for_each**

- **mismatch**

- **equal** — comparing ranges

- **is_permutation**

- `copy, copy_n, copy_if, copy_backward`

- `move, move_backward`

- `swap_ranges`

- `transform`

- `replace, replace_if, replace_copy, replace_copy_if`

- `fill, fill_n`

- `generate, generate_n`

- `remove, remove_if, remove_copy, remove_copy_if`

- `unique, unique_copy`

- `reverse, reverse_copy`

- `rotate, rotate_copy`

- `shuffle`

- `is_partitioned, partition, stable_partition, partition_copy, partition_point`

- **sort, stable_sort, partial_sort, partial_sort_copy**

- **is_sorted**

- **nth_element**

- **lower_bound, upper_bound, equal_range**

- **binary_search**

- **merge, inplace_merge**

- **includes, set_union, set_intersection, set_difference, set_symmetric_difference**

- **push_heap, pop_heap, make_heap, sort_heap**

- **is_heap_until, is_heap**

- **min, max, minmax**

- **min_element, max_element, minmax_element**

- **next_permutation, prev_permutation**

- **Additional algorithms might come in the future**

- **Some algorithms can be parameterized by execution policies**

  - New first parameter of type ExecutionPolicy

  - `std::execution::sequenced_policy` (`std::execution::seq`)

    - Executed in the calling thread

  - `std::execution::parallel_policy` (`std::execution::par`)

    - May be executed in another thread

  - `std::execution::parallel_unsequenced_policy` (`std::execution::par_unseq`)

    - May be executed in other threads and vectorized

- **Use algorithms whenever they are more expressive and shorter to use than loops**

- **Function, lambdas and functors can be passed as arguments of the algorithm's actions and predicates**

- **Don't invalidate your iterators as a side-effect**

- **Ensure you get your memory management right for an algorithm's output**

- **Familiarize yourself with the available algorithms**

```
#include <boost/iterator/counting_iterator.hpp>
#include <boost/iterator/filter_iterator.hpp>
#include <boost/iterator/transform_iterator.hpp>
```

● **Several pre-defined adapters with factory functions, for example**

  ▪ Counting

  ▪ Filtering

  ▪ Transforming